Identifying executable plans

Tania Bedrax-Weiss* Jeremy D. Frank
Ari K. Jónsson† Conor McGann*
NASA Ames Research Center, MS 269-2
Moffett Field, CA 94035-1000,
{tania,frank,jonsson,cmcgann}@email.arc.nasa.gov

Abstract

Generating plans for execution imposes a different set of requirements on the planning process than those imposed by planning alone. In highly unpredictable execution environments, a fully-grounded plan may become inconsistent frequently when the world fails to behave as expected. *Intelligent execution* permits making decisions when the most up-to-date information is available, ensuring fewer failures. Planning should acknowledge the capabilities of the execution system, both to ensure robust execution in the face of uncertainty, which also relieves the planner of the burden of making premature commitments. We present Plan Identification Functions (PIFs), which formalize what it means for a plan to be executable, and are used in conjunction with a complete model of system behavior to halt the planning process when an executable plan is found. We describe the implementation of plan identification functions for a temporal, constraint-based planner. This particular implementation allows the description of many different plan identification functions.

Introduction

Planning has been an important problem in Artifcial Intelligence, but *plan execution* is a problem that AI has often overlooked. In fairly simplistic and highly-predictable environments, the planning system can hand a fully-grounded plan to the execution system and the execution system will be able to execute it flawlessly. However, in unpredictable environments, a fully grounded plan will often result in execution failures. Additionally, it is inefficient for the planning process to commit to decisions in advance that are likely to be invalidated during execution. One way to avoid these problems is to have an *intelligent execution system* that is able to "fill in the blanks" given a plan that is not fully grounded. Intelligent execution may range in complexity from fairly simple computations to a process resembling full-blown planning. Depending on the characteristics of the execution environment, the best plan to hand to the execution system will contain more or less commitment and information.

AI solutions for planning and plan execution often use declarative models to describe the domain of interest. The planning system typically uses an abstract, long-term model and the execution system typically uses a concrete, short-term model. In most systems that deal with planning and execution, the language used in the declarative model for planning is different than the language used in the execution model. This approach enforces a rigid separation between the planning model and the execution model. The execution system and the planning system have to agree on the semantics of the plan, and having two separate models requires the system designer to replicate the information contained in the planning model in the execution model. Since much of the knowledge may be shared between the planning and execution systems, this produces a model synchronization problem. Furthermore, if the execution system wants to relay information back to the planner, in this scenario, the information has to be translated. Not using the same language may be detrimental because synchronization and translation may introduce errors and increase model engineering effort.

If both the execution and planning systems have a common language, then information from and to the executive can flow seamlessly. Furthermore, changes in the planning model are then automatically propagated to the execution model. In addition, it is easier to modify the boundary between planning and execution systems in such systems. Even if a single model for planning and execution is not available, there are still good reasons for making it easy to vary the boundary between planning and execution systems using a single model of system behavior. Many candidate execution systems may be available; identifying the right one requires a planning model that can be used with any one of the possible execution systems.

An *executable plan* is a plan that the execution system can make any remaining decisions and then translate the result into commands that can be sent to the hardware with which the system is interacting. Based on this notion, the "executability" of a plan depends directly on the capabilities of the execution system. For a general model of system behavior, we would like to specify which parts of that model describe the commit-

[1]QSS Group, Inc.
[2]Research Institute for Advanced Computer Science

ments that should be made by the execution system. In this paper we provide the formalism and implementation of Plan Identification Functions PIFs . These functions exactly characterize the executable plans in a model that is shared between the planner and the executive. Thus, they serve to separate the duties of planning by partitioning the planning problem into a set of commitments that are made in advance, and a set that are interleaved with plan execution.

We will proceed by presenting a simple example to clarify some of these notions.

A Simple Example

To examine the issues involved in generating plans for execution, let us consider a simple spacecraft that can slew (i.e, turn to different orientations), take pictures, and download pictures to Earth.

A plan request for this spacecraft might consist of a set of picture requests, and then a request for downloading some or all of these pictures to Earth. The planning process would generate an "executable plan" that achieved those goals. The execution agent would then execute the plan by thrusting to rotate the spacecraft, activating camera components, and transmitting data.

A traditional approach to this problem would be based on separating the planning from execution at some specific level of abstraction. For example, the planning process might involve generating slew actions, orientation maintenance actions, picture-taking actions, and download actions. The complete plan, at that level of abstraction, would then be executed by breaking each high-level action down into specific commands that together perform the action. The slew actions, for example, would be broken down into engine warmup, thruster firing, wait, opposite thruster firing, and then stabilization. The sequence of execution commands would have to fit within the time allocated to the slew action.

If the execution system is sophisticated, it could determine how fast to slew, since the slewing rate is controlled by the time spent firing the engines at the start and end of the slew. Slower slew rates typically save fuel. The execution system might also be able to determine when to start activities that have some temporal flexibility. Rather than waiting until the next action start time, which is bound to be later rather than earlier, to provide a safety margin, the execution system could determine that all earlier actions have been completed, that the action in question can start as early as a given time, and then start that action at that time.

In the Remote Agent Experiment (JMM+00; MNP+98), the planning process built a partial plan where certain temporal decisions were left to the execution engine. These temporal decisions were limited to those that together formed a simple dispatchable network (MMT98). However, as noted in (JMM+00), the notion of retaining flexibility in the executed plan can be generalized to an arbitrary set of decisions.

Models of System Behavior

The system model defines the set of possible states and actions, along with rules that specify allowed and forbidden relations between actions and states. The states and actions are defined by predicates like `pointingAt(object)` and `slewFromTo(obj1, obj2)`. We define a *partial plan* as a collection of temporal predicate statements, based on a given model. We assume that the parameter set for each predicate doesn't have to be fully grounded. This notion naturally supports more flexible planning paradigms such as those used in constraint-based planning systems.

The rules in the system model specify, directly or indirectly, the conditions for a state or action being in the plan:

- which instantiations of state/action predicates are valid
- for each action/state, what other states or actions must exist in the plan to support it
- the temporal and parametric relations between states and actions

For example, a model for our simple spacecraft would state that the predicates include `pointingAt`, `slewingFromTo`, `takePicture`, `download`. Legal instantiations of `slewingFromTo` are limited to those where the origin and destinations are not the same, and `takePicture` instantiations are limited to objects and times where an overly bright object, like the sun, is not in the frame. There are a number of relations between predicates in this model; for example, a `takePicture` must be done within a period where the spacecraft is pointing at the object in question. This means that there must be a `pointingAt` predicate in the plan, and that furthermore, the appropriate `pointingAt` must have the same object value as the `takePicture`, must start no later than the `takePicture` and must end no earlier than the `takePicture`.

It should be noted that this core approach covers basic STRIPS-like descriptions, where states are preserved by the frame axioms and action definitions define state changes. It also covers constraint-based approaches, with and without an explicit representation of time or resources. Furthermore, in STRIPS, the initial state and goal statements are simply part of a partial plan that must be extended to include the actions necessary to go from the initial state to a goal state. Since both planning and execution use the same model, the plan is semantically meaningful to both the planner and the executive [1].

The rest of the paper is organized as follows. We first provide a formal definition of PIFs and characterize some useful properties of PIFs . We then describe an implementation of PIFs in a planning framework called EUROPA. We identify some important implementation details that arise when implementing PIFs

[1] This idea is borrowed from (MDF+02), which we describe later in the paper.

in this framework. We then conclude and discuss several open issues.

Plan Identification

We now turn our attention to formally defining the concepts related to general plan identification. We begin with a general and expressive approach to planning, which supports arbitrary variables, quantitative temporal relations, arbitrary constraints, and expressive activity-state rules.

Constraint-based planning

In order to address realistic problems, a planning paradigm must support actions and states with temporal extent, complex relations among action and state arguments, as well as complex model rules about conditions and effects of actions and states. In recent years, different approaches have been proposed for moving away from the classic STRIPS paradigm, and towards more realistic approaches that incorporate explicit representations of time and resources. These approaches fall into a broad category called *Constraint-Based Planning* (CBP) (SFJ00).

The basic idea behind CBP is to use variables to represent all aspects of states and actions, and to use constraints to enforce relations between those variables. The basic element in constraint-based planning is an interval. An *interval* is simply a predicate holding over a period of time. The start and end of the interval and the parameters of the predicate are described by variables. More formally, an *interval* is a tuple, (p, X, s, e), where B is a predicate name, X is a vector of variables defining the arguments to the predicate, and s and e are temporal variables, defining the start and end of the interval.

A *planning domain* is defined by the set of interval types, and a set of configuration rules. A *configuration rule* is a generalization of the notion of preconditions and effects. Instead of specifying only state values before and after an action, a configuration rule can specify arbitrary temporal relations specifying how actions and states must relate in a valid plan. This means that a configuration rule can specify that whenever an attribute is assigned an interval of a certain kind, other intervals must exist in the plan, such that specific constraints are satisfed. In addition to temporal constraints, configuration rules can specify other constraints amongst the parameters of the actions and states.

In our spacecraft example, consider a `takePicture(x)` interval, I, for the camera attribute. A configuration rule might specify that there must be a `pointingAt(y)` interval, J, such that $x = y$, the start of J is at least 10 seconds before the start of I, and the end of J is no earlier than the end of I. Notice that the latter constraint in the rule is not strictly a precondition or an effect, but only involves temporal constraints.

In constraint-based planning, partial and complete plans are represented as networks of intervals. The connections between intervals in such a network are defined by the configuration rules.

Partial plans and completions

In CBP, a *partial plan* consists of a set of intervals and a set of constraints among the variables representing those intervals.

A partial plan P is *valid*, if for every applicable configuration rule, all the intervals and constraints required by those rules are in P. A partial plan P is *instantiated*, if each variable has been given a single value. A partial plan P is *consistent* if none of the constraints in the plan are violated and *inconsistent* otherwise.

A *planning problem* is simply a partial plan. This notion generalizes the very restrictive STRIPS notion of only specifying an initial state and a set of goals. The notion of a planning problem as a partial plan allows specific actions as goals, supports the specification of maintenance goals, makes it easy to define exogonous events, and much more. Planners can modify plans in two ways. A *restriction* is defined as the binding of a variable or the addition of a constraint. A *relaxation* is defined as the unbinding of a variable or the removal of a constraint. An *extension* of a given partial plan, P, is a plan Q such that each interval in Q can be mapped to a compatible interval in P, and each constraint in P is in Q. Thus, restricting a plan P results in an extension Q, and relaxing a plan Q' results in a plan P' such that Q' is an extension of P'.

A partial plan, Q is *complete* if every interval is instantiated and the plan is valid. Q is a *completion* of every relaxation of Q. We say that a problem instance P has a *solution* if it has a consistent completion Q.

The strictest notion of solving a planning problem P is to find a consistent completion of P. However, a more general notion is much more useful when it comes to planning for execution agents that are not completely trivial in complexity. In essence, solving a planning problem P for an intelligent execution agent involves finding a consistent extension Q that can be executed by the given execution agent. We now turn our attention to a general formulation of such a notion.

Plan identification functions

Consider a partial plan encountered during the course of planning. We would like a declarative description of the set of plans that can be accepted for execution by the execution system. This is the notion of *plan identification functions* (PIFs). The basic idea is to have a mapping that indicates whether or not a partial plan is suitable for a given execution engine or not.

Identifying inconsistency is a natural complement to plan identification. Consider a partial plan that has no valid completions. In a sense, the partial plan is a dead-end. However, it is computationally expensive to determine whether any given partial plan is inconsistent or not. Consequently, it is useful to think of a

"consistency identification function" that maps partial plans to T, F, or ?, where the T value indicates that the plan is consistent, F indicates it is inconsistent, and ? indicates that the consistency of the plan is not known.

The original notion of a PIF appeared in (JMM+00). This definition combined the notion of consistency with executability, and used three return values, T, F and ? A return value of F indicated that the plan violated some constraint, i.e. no extension of the plan was consistent, which forced the planner to backtrack. A return value of T indicated that all intervals were valid and consistent according to a set of applicable configuration rules, and thus planning was complete. A return value of ? indicated that the plan was consistent but not valid, and thus the planner had to continue searching for an extension.

In this paper, we define a more relaxed notion of a plan identification function. A PIF maps partial plans to the values Y and N. A return value of Y indicates a plan is executable and a return value of N indicates a plan is not executable. Keeping the definitions as general as possible, we do not pose any more restrictions on the evaluations of partial plans. For example, it is possible that plan execution systems may be based on technology that works in the space of inconsistent plans, and thus we want to be able to specify PIFs that are able to pass inconsistent plans to the execution system. We therefore look at specific characteristics of PIFs that are desirable in certain cases.

Characteristics of plan identification functions

Regardless of the impact of execution, we assume that a partial plan P must be consistent when the plan execution commences. We also assume that the instructions the execution system issues to the underlying hardware are based on fully instantiated plans. If either of these conditions is not satisfied, then the plan execution system must be able to come up with a consistent complete plan.

The simplest question one can ask of a plan handed to the execution system is whether it has a consistent completion. In most (but not all) cases, such correctness would be a crucial characteristic of an executable plan.

A PIF , i, enforces *correctness* if, for any partial plan P, such that $f(P) = \mathrm{y}$, P has at least one consistent completion.

The next question is how much work needs to be done by an execution engine to find a consistent completion in different circumstances. This is a particularly interesting question if uncertainty during execution is taken into account.

A PIF , i, enforces *solvability* if, for any partial plan P, such that $i(P) = \mathrm{y}$, all extensions of P are complete and consistent.

It is often difficult to find plans satisfying the above property without finding a complete plan to begin with. As such, it is useful to identify PIFs that return partial

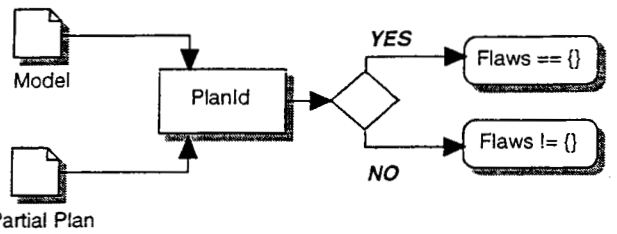

Figure 1: The Plan Identification Function as a Flaw Filter

plans that require only a bounded amount of time to solve. In cases where a PIF returns inconsistent plans to the executive, it is also useful to characterize the amount of time that search in the infeasible space is performed. Because of the different search procedures required we provide two definitions below.

We first want to formally characterize an executive that can efficiently ensure a completion of a consistent plan can be found, if one exists. A PIF , i, enforces $O(f(n))$ *solvability* if, for any consistent partial plan P satisfying $i(P) = \mathrm{y}$, then in time $O(f(|P|))$ either a consistent completion of P can be found or it can be shown that no consistent completion of P exists.

On the other hand, if the executive is handed an inconsistent plan, the problem is somewhat different. In this case, no completion is possible. Instead, we must find a completion of the original planning problem that the planner attempted to solve. Suppose the original planning problem is R. A PIF , i, enforces $O(f(n))$ *transformability* if, for any partial plan P satisfying $i(P) = \mathrm{y}$, then in time $O(f(|P|))$ a series of transformations of P resulting in a Q, a consistent completion R, can be found, or it can be shown that no such completion exists.

Examples

In the Remote Agent Planner, the PIF accepted only consistent and valid plans where all parameter variables had been assigned specific values, but tolerated unassigned temporal variables forming a dispatchable simple termporal network (JMM+00). The restriction that the resulting temporal network be dispatchable made the PIF correct and provided a linear bound on how much time it would take the execution engine to complete a given plan. This is an $O(n)$ solvable PIF .

Recent techniques have extended the ability of execution systems to handle uncertainty in temporal quantities. In particular, (MMV01) presents an algorithm that can detect when a temporal network with uncertainty can be executed without failure, and (MM01) presents an algorithm for executing such networks in polynomial time. Thus, we can write PIFs for such problems that are $O(f(|P|))$, where f is a polynomial.

From Plan Identification to Flaws

We have formally defined the PIF as a function from a plan and a model to an answer of either `Y` or `N`. In practical applications, however, a planner using the PIF would like further indication of what is wrong with the plan when the answer is `N`. A more useful notion of a PIF , depicted in Figure 1, is one that given a plan and a model, returns a set of possible plan modifications if the answer is `N` and the empty set if the answer is `Y`. We refer to the set of plan modifications as flaws.

A *flaw* is a modification to a partial plan, either a restriction or a relaxation. Let $\mathcal{F}(P)$ be the set of flaws derived from plan P. We can now redefine a PIF as a mapping: $\mathcal{F}(P) \rightarrow 2^{\mathcal{F}(P)}$. That is, the PIF identifies a (possibly proper) subset of the flaws that define the set of plan modifications that a planner can make.

The set of flaws can be defined in different ways depending on how the planner conducts search. For example, suppose the partial plan has valid extensions. Then the set of flaws might consist only of the set of restrictions that a planner can impose in its search for a completion that satisfies the PIF . However, a planner such as ASPEN (FRCY97) can benefit from flaws that are restrictions or relaxations, since it can search the space of infeasible solutions.

Notice that it could require exponential space to define very complex plan identification functions if it were necessary to enumerate all the possible sets of flaws and the mapping that applied for each of those flaw sets. Practicality dictates that we have a concise manner of both expressing and evaluating PIFs .

An Implementation of Plan Identification

We have implemented the notion of a PIF as a flaw filter in the context of the *Constraint-based Attribute and Interval Planning* framework (FJ03) (CAIP), in the system called EUROPA (Extensible Universal Remote Operations Planning Architecture). In this section we first give an overview of EUROPA, then describe the PIF implementation. Further details on EUROPA implementation can be found in (FJ03); in this section, we focus on those aspects that are most relevant to PIFs .

EUROPA Overview

CAIP is an extension of CBP. Like the basic constraint-based planning paradigm, intervals provide the basic representation of actions with durations and states with temporal extent. The key addition is in the notion of an attribute. An attribute represents some system, subsystem or other aspect of the domain for which planning is being done. An attribute can only take on one value at a given time, so attributes enforce a mutual exclusion relation among intervals that are assigned to the same attribute. In addition, each interval must be placed on an attribute. This requirement enforces mutual exclusion among all intervals.

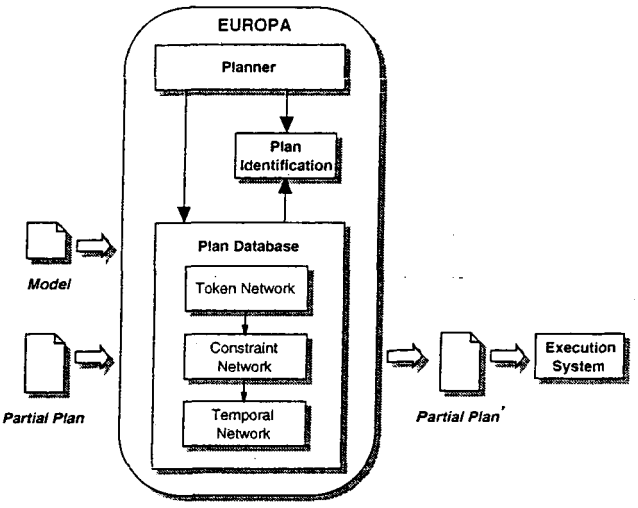

Figure 2: Planning For Execution With EUROPA

In our spacecraft example, the state of the camera might be one attribute. When a picture is being taken, the camera takes on the state `takePicture`, otherwise, the state is `cameraIdle` or `slewingFromTo`. Similarly, the attitude of the spaccraft could be another attribute, whose values are instantations of `slewingFromTo` and `pointingAt`.

A partial plan in EUROPA consists of a mapping of attributes to sequences of intervals, a set of free intervals, and a set of constraints on variables in the given intervals. Free intervals are intervals that have not been sequenced on attributes yet. We assume for simplicity that the set of flaws of a partial plan is comprised of free intervals and unbound variables [2]. A plan identification function, then, takes the set of free intervals and unbound variables in the plan database and returns a subset of these in response to a query from the planner.

Figure 2 shows the overall architecture of EUROPA in the context of planning for execution. The system is composed of the following modules: a *planner*, a *plan database*, and a *plan identification* module. Planning begins with a partial plan and a domain model. The plan database is initialized with the partial plan and the model. During planning, a planner can query the plan database through the plan identification module for flaws in the initial partial plan. Flaws are defined in terms of the model configuration rules and the PIF acts as a filter. If no flaws remain and the plan is consistent, the planner concludes that a plan has been found. If flaws remain, however, the planner makes commitments to resolve the remaining flaws by updating the plan database. The planner thus alternates between asking the plan identification module for flaws and updating the plan database until a plan that satisfies the model and the PIF is found.

[2]Note that this set of flaws is only useful for planners that search in feasible space, but EUROPA can support other flaws as well.

In the EUROPA plan database, predicate arguments, timepoints, and attributes of an interval are represented as variables. Configuration rules impose constraints on the values these variables can take. The plan database manages the variables and constraints and makes use of a temporal network to maintain consistency between temporal variables and the temporal relationships imposed by the configuration rules. The plan database also uses a constraint network to maintain consistency among all other variables and constraints.

Consider the following simple model of the spacecraft domain. The first half of the model specifies the intervals that can appear on each attribute, and the second half specifies the configuration rules. We use the simple temporal relations of Allen's Algebra to specify constraints between the timepoints of required intervals. We also assume that parameters of different intervals with the same variable name require the parameters to take on the same value.

```
Attitude:{pointAt(object), turnTo(object)}
Camera:{off(), ready(), takePic(object)}
Take-Picture(B) → met-by ready()
Take-Picture(B) → contained-by pointAt(B)
ready() → met-by off()
pointAt(B) → met-by turnTo(B)
```

Figure 3: A simple model of the spacecraft domain.

EUROPA enforces configuration rules by means of the following *plan invariant*: whenever a plan modification results in a change to the set of plan completions, the intervals in the plan are updated. New intervals are added as free intervals. In the case of relaxations, some intervals that were part of the plan may no longer be justified, and if so, the intervals and all associated variables and constraints are removed. In the case of restrictions, new intervals, variables and constraints may be needed in the plan, and if so, they are added.

Figure 4 shows a plan fragment based on the simple model. The `Camera` attribute is initially turned off, then it is ready, and then it is taking a picture. While the `Camera` is taking a picture of the object, the `Attitude` is pointing at the object. Notice that there are two free intervals, one with predicate `turnTo` and one with predicate `pointAt`. The free interval `turnTo` was generated by the plan invariant, while the free interval `pointAt` was part of the initial problem instance.

Consider the interval `pointAt`(A) which is inserted on the `Attitude` attribute. In this case, the rule `pointAt`(B) → `met-by turnTo`(B) means that if a `pointAt`(B) interval exists in a plan, then a `turnTo`(B) must precede the `pointAt`(B). The presence of the `pointAt`(A) interval forces the addition of the free `turnTo`(C) interval due to the plan invariant. Similarly, if the `pointAt`(A) interval is removed from the `Attitude` attribute, then the free interval `TurnTo`(C) is no longer justified, and is removed from the plan.

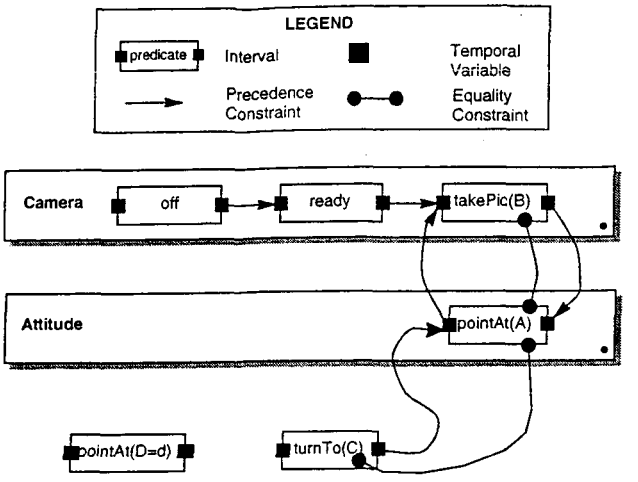

Figure 4: A simple partial plan for the model described in Figure 3

In EUROPA, the parameter equivalence is handled by creating a new variable for the required `turnTo` interval and posting an equivalence constraint between the parameters. In this example, the parameter C of the `turnTo` predicates has been equated with the parameter A of the sequenced `pointAt()` interval. Finally, we note that C has not been bound to any particular value, while parameter D of the other `turnTo` has been bound to value d.

Plan Identification in EUROPA

In EUROPA, the PlanId function is implemented as a filtering operation on the set of flaws in the plan database. To support this, the system must provide capabilities to:

1. obtain access to the set of flaws in the plan database;
2. define a filter expressing criteria for including or excluding a flaw;
3. obtain a set of filtered flaws by applying such a filter.

These capabilities are accomplished by providing:

1. a flaw storage mechanism, referred to as the `FlawCache`, which keeps the set of flaws in the plan database synchronized with changes made through explicit commitments by the planner or derived through inference.
2. a highly customizable filtering structure which allows pre-defined conditions and/or new custom conditions to be seamlessly integrated in a single filter.
3. a flaw querying facility which handles all access to the `FlawCache` and applies filtering criteria defined by the planner.

The remainder of this section describes in more detail the framework developed to achieve this in an efficient and customizable manner.

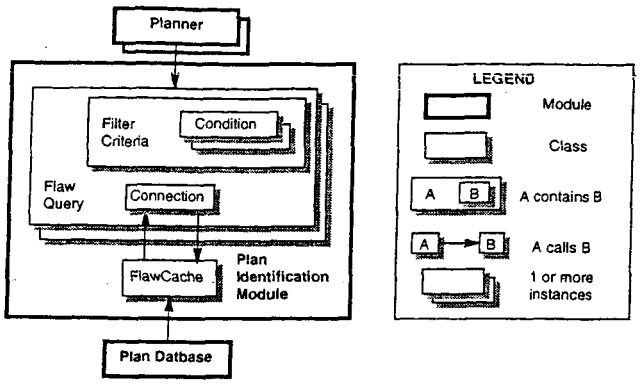

Figure 5: Class Diagram of the PlanId Framework

Framework Class Diagram

Figure 5 presents the internal details of the *PlanId* module referenced in Figure 2. The `PlanDatabase` generates events indicating changes to intervals and variables when the plan invariant is invoked. These events are received by the `FlawCache` and used to maintain the set of all flaws in the system, i.e. all free intervals and unbound variables. Events indicating a restriction may cause a flaw to be removed from the `FlawCache` e.g. inserting a free interval or assigning a value to an unbound variable. Events indicating a relaxation may cause a flaw to be inserted into the `FlawCache` e.g. relaxing to domain of a variable or freeing an inserted interval.

A planner creates a `FlawQuery` at the beginning of the planning process. It is by means of a `FlawQuery` that a planner obtains the relevant subset of flaws as indicated by a filter. Planner-specified filters are defined in a `FilterCriteria` object, which is just a collection of `Conditions`. Each `FlawQuery` has exactly one `FilterCriteria` instance, provided to it during construction. `Condition` objects provide the customization necessary for planners to filter out flaws they wish to ignore. For a `Flaw` in the `FlawCache` to be returned by a `FlawQuery`, *all* `Conditions` must be satisfied.

In order to gain access to the set of flaws and the set of flaw changes, each `FlawQuery` establishes a `Connection` with the `FlawCache`. A `Connection` provides access to *all* flaws in the `FlawCache`. A `Connection` also provides a location to store information on changes in the `FlawCache` since the the `FlawQuery` was last queried. Notifications of changes in the contents of the `FlawCache`, i.e. flaws inserted or removed, are pushed to each connection from the `FlawCache` as the latter is synchronized with the `PlanDatabase`.

This architecture provides a number of useful features. First, the FlawCache can support many connections at once, enabling it to provide flaws to many planners. Second, a wide variety of simple conditions are provided, enabling a very large number of different PIFs to be expressible. Third, it is very straightforward to develop additional conditions making the approach very extendible. Finally, emphasis on lazy evaluation and event-based synchronization leads to efficient implementation.

```
while(done==false)
   if (isConsistent())
     filteredFlaws=getFlawsFromQuery()
     if (filteredFlaws.isEmpty()==false)
       nextFlaw = choose(filteredFlaws)
       resolve(nextFlaw)
     else done=true
   else... // rest of the algorithm omitted
end while
```

Figure 6: Planning with Flaw Queries.

Step 1:
FlawCache=$\{A, B, C$,`pointAt`$(D = d)$,`turnTo`$(C)\}$
FilteredFlaws:$\{A, B, C$,`pointAt`$(D = d)\}$
nextFlaw: `pointAt`$(D = d)$
Step 2:
FlawCache=$\{A, B, C, E$,`turnTo`(C),`turnTo`$(E)\}$
FilteredFlaws:$\{A, B, C, E\}$
nextFlaw: E
Step 3:
FlawCache=$\{A, B, C\}$`turnTo`$(E = d)$,`turnTo`$(C)\}$
FilteredFlaws:$\{A, B, C\}$
nextFlaw: A

Figure 7: Evolution of the flaws for the partial plan in Figure 4.

To see how the flaw filtering works, consider the sample partial plan shown in Figure 4. There are five flaws: the variables A, B and C, the `turnTo`(C) interval and the `pointAt`$(D = d)$ interval; the FlawCache has these five flaws. Now suppose that the PIF filters out intervals with predicate `turnTo`. Then the set of filtered flaws consists of the three variables and the `pointAt`$(D = d)$ interval.

The basic loop of a planner is similar to the fragment presented in Figure 6. At each step, the planner requests the filtered flaws. Once the flaws are retrieved, the planner uses some criteria to select a flaw, then uses another criteria to resolve the flaws. Application of the plan invariant and propagation of variable changes in the constraint network result in updates to the FlawCache. Subsequent queries to the FlawQuery will return a new set of flaws that accounts for these updates and the filtering of these flaws by the PIF .

To see this process in action, let us consider a few steps of planning given the partial plan and PIF that we have described. This process is shown in Figure 7. Let us assume that *choose* selects flaws according to some arbitrary order. Also suppose that *resolve* performs an

insertion for free intervals or a variable assignment for unbound variables. After inserting `pointAt`$(D = d)$ we see that the plan invariant ensures the creation of a `turnTo`(E) interval. The FlawQuery, however, indicates that the set of filtered flaws at step 2 only includes the variables A, B, C, E. At the next step, choose() returns flaw E; there is only one possible value, d, and thus the plan invariant doesn't lead to the creation of any new variables or intervals.

Such a simple filter could be achieved with a single condition which would check the predicate name of each interval flaw and exclude it if it matched the name `turnTo`.

EUROPA Plan Identification Function Capabilities

EUROPA's PIF framework supports the following conditions, among others:

- Interval predicate filtering - filters all intervals of a particular predicate.
- Interval variable filtering - filters selected variable of all intervals with a particular predicate.
- Attribute filtering - filters all intervals and all variables of all intervals from a particular attribute.
- Temporal filtering - filters intervals according to a variety of temporal specifications. One example is a filter for intervals guaranteed not to happen within a temporal extent (a horizon filter).

In practice, different applications will impose different requirements on plan executives. In an execution environment that is uncertain, there are advantages to not fully specifying a plan. To be robust against uncertainty in the execution environment a plan must be flexible. For instance, if decisions cannot be determined in advance because the way in which they are made depends on factors that are only determined at execution time, it may be advantageous to leave these decisions up to the execution system. The PIF framework allows considerable latitude in defining the capabilities of execution systems, and thus enables the planning technology to be more widely useful. However, it also provides considerable flexibility within a single application. Engineers can design different PIFs and analyze the resulting performance of the integrated planning and execution system, and choose the PIF that works best.

An execution system will typically only care about the plan developing inside the current execution window. If this execution system is implemented as a planner, a PIF could be used to focus the planning effort on that execution window only using the horizon filter. Such a PIF would contain a horizon condition that would specify, for each free interval and each unbound variable, whether it falls within the horizon or not. A free interval falls within the horizon if its start time and end time variable domains include the horizon timepoints. An unbound variable will fall inside the horizon if it belongs to one of the intervals that falls within the horizon.

The time at which an event actually occurs is usually different from the planned time. This difference can sometimes prove costly since it may cause some assumptions that were made in the planning stage to fail. In EUROPA, the temporal network is implemented as a Simple Temporal Network (DMP91). Simple Temporal Networks guarantee that if the network is consistent, an appropriate set of bindings of the temporal variables can be found in polynomial time. Thus, if the temporal network is consistent, no further commitments on time have to be made during planing. This is assuming that the executive is intelligent enough to be able to find this appropriate set. A PIF provides the means to define this flexibility if it implements a condition that filters temporal variable flaws. If no temporal variable flaws are passed on to the planner, these decisions will remain unbound (though constrained by the temporal network) until execution time.

Overcommitment at planning time may prove costly in other ways. In cases where a plan consists of high-level and low-level tasks, the low-level task expansion of the high-level tasks may depend highly on when the tasks get executed. In such cases, it is convenient to let the execution system map high-level tasks into low-level tasks during execution. This frees the planner from generating low-level tasks, and allows the executive to choose the low-level tasks that best fit the actual execution. In EUROPA, high-level and low-level tasks can be placed on separate attributes. A PIF provides the means to define this flexibility by implementing a condition that filters flaws depending on whether they are allowed to be placed on high-level attributes or not. A free interval can be placed on a high-level attribute if it belongs to the set of allowable states of that attribute. An unbound variable belongs to the high-level attribute if it belongs to an interval that can be placed on the attribute.

There are instances of planning for execution when some planning decisions inside one execution cycle may determine what will happen in a future execution cycle. These commitments are sometimes unnecessary, especially in uncontrolled execution environments. These planning decisions may manifest themselves as particular predicate logic statements or as arguments to predicate logic statements. A PIF povides the means to delay commitment on these predicates or variables by implementing a condition that filters flaws on whether they are based on these predicates or variables.

Complexity Analysis

In the simplest implementation, one could omit the `FlawCache` and `Connection` infrastructure. Resolving a query would be accomplished by iterating over all intervals and variables in the plan database and for each, applying the filter to test for inclusion or exclusion. This would result in a worst-case time-complexity given by $(N_v + N_i) * N_c * C_c$ where N_v is the number of vari-

ables, N_i is the number of intervals, N_c is the number of conditions in the filter, and C_c is the average cost of evaluating a condition. [3]

Since the points of greatest cost are in the evaluation of conditions, we seek to reduce the execution of condition tests. This is accomplished in a numnber of ways:

1. The last set of filtered flaws are cached in each `FlawQuery`.
2. The current set of flaws in the plan database are cached in the `FlawCache`.
3. Each cache is maintained through notifications of changes.
4. Conditions may be ordered to fail fast, based on the characteristcis of each problem.
5. The FlawQuery is updated only when the planner consults it for the latest set of flaws. Thus, the queries are only run on the set of flaws that were added since the last query.

The resulting worst-case cost of a query is approximated by: $N_+ * N_c * C_c$ where N_+ is the number of flaws inserted into the flaw cache since the last query.[4] The approximation omits the cost of caching events during synchronization of the FlawCache and the PlanDatabase. This is reasonable since the costs of caching are much less than the cost of evaluating the conditions over all insertions. Notice that we do not need to worry about flaws that are removed from the cache, since they aren't returned to the planner in any case.

Related Work

A wide variety of agent architectures have been designed to support both planning and execution. We will not describe all aspects of these systems here. We will describe how these systems characterize the boundary between planning and execution, and compare it to the approach we have described here.

Many integrated planning and plan execution frameworks define a fixed boundary between their components. These systems also use different modeling languages, in some cases with different semantics, and thus have potential problems with model synchronization. Finally, these systems do not have a crisp declarative characterization of the boundary between the components. Examples of integrated planning and plan execution systems in this category are O-Plan (TDK94), 3T (BFG+97), and Propice-Plan (DI99).

Cypress (WMLW95) is a planning and plan execution framework designed for a variety of applications, including military operations. Cypress is a loosely coupled integration of the SIPE planner, the PRS reactive execution system, and Gister-CL system for reasoning under uncertainty. Cypress enables human intervention during planning and plan execution. Cypress uses the ACT representation to model both planning and execution. The boundary between SIPE and PRS is flexible, as PRS can invoke SIPE to handle run-time plan failures. However, there is no facility in Cypress to describe the boundary between the planner and plan execution in a declarative way.

The Remote Agent (RA) (MNP+98; JMM+00) is an agent architecture for spacecraft control that was used in a 2-day experiment of an autonomous probe. The RA consisted of a planner, a plan execution system, and a mode identification and reconfiguration system. The RA planner built plans that were temporally flexible so that the plan execution system could decide on-the-fly which tasks to start and end (MMT98). This represented a significant advance at the time; however, other applications using the RA could not use any other divide between planning and execution. Furthermore, the three components of the system used different modeling languages with different semantics, requiring considerable effort to ensure model synchronization.

IDEA (MDF+02; DLM03) is an agent architecture designed to overcome shortcomings in the RA approach to agent modeling. IDEA provides a simple virtual machine that supports plan execution, consisting of a model, plan database, plan runner, and reactive planner. The job of the reactive planner component of an IDEA agent is to ensure that a "locally executable" plan is returned. Thus, a crucial task is to define the scope of the Reactive Planner's job. The PIF is a natural way to focus on those parts of the model that must be addressed by the Reactive Planner. IDEA also supports many planners operating on the same plan database, and thus the same model. PIFs are a natural way to define the scope of these various planners in order to ensure that planners do not step on each others' toes. IDEA also supports multi-agent architectures using inter-agent communication. The original notion of IDEA is to separate models for each agent; these models are intended to be written in the same language and share components. Partial plans serve as the medium by which planners communicate with the executive, as well as the medium by which IDEA agents communicate with each other. However, the PIF can (in principle) be used to simply divide up the model amongst the agents in a similar manner to the way it divides up models amongst planners; the crucial problem to solve is dividing plan databases efficiently among the IDEA agents.

Conclusions and Future Work

We have described plan identification functions as a way of circumscribing the planning problem that *must* be solved in order to create an executable plan. PIFs have the advantage of enabling a single model to characterize both the planning problem and the plan execution problem. They also enable easy characterization of the boundary between planning and plan execution, even

[3] In practice only some of the conditions will be executed since we discard the flaw after the first condition fails.

[4] $N_+ << (N_i + N_v)$ since there are relatively few flaw insertions resulting from each planner commitment.

in cases where different models for planning and execution are used. They also provide considerable flexibility, as they allow the boundary between planning and execution to be adjusted. We have described the implementation of the PIF framework of EUROPA, and shown how it can be used to implement many PIFs for different type of plan execution systems.

We have implicitly assumed that a single model of system behavior can be written, so that PIFs can be used to separate the part of system behavior that pertains to the execution system. The IDEA project (MDF+02) is pursuing this notion, but it remains to be seen how the concepts extend to more sophisticated planning and control architectures.

We have described one way of using PIFs to divide a model amongst many planners. This approach does not address important architectural issues of mutli-agent access to a shared plan representation. It also doesn't address the issue of how to structure the plan representations used by planners and executives. The efficient implementation of PIFs may be impacted by this architecture.

Note that while the plan that is passed to the executive may define a set of plan completions, there is no reason to assume that the executive chooses one of these completions, and in fact no way to characterize the actions of the executive in a declarative way.

PIFs can be used for more than just separating planning from the execution system. One can also imagine partitioning the planning problem into many different problems using a collection of PIF functions.

References

R. Bonasso, R. Firby, E. Gat, D. Kortenkamp, D. Miller, and M. Slack. Experienences with an architecture for intelligent, reactive agents. *Journal of Experimental and Theoretical Artificial Ingelligence*, 9(2), 1997.

O. Despouys and F. Ingrand. Propice-plan: Towards a unified framework for planning and execution. In *Proceedings of the 5th European Conference on Planning*, 1999.

M. Dias, S. Lemai, and N. Muscettola. A real-time rover executive based on model-based reactive planning. In *Proceedings of the International Conference on Robotics and Automation*, 2003.

R. Dechter, I. Meiri, and J. Pearl. Temporal constraint networks. *Artificial Intelligence*, 49:61–94, 1991.

J. Frank and A. Jónsson. Constraint based attribute and interval planning. *Journal of Constraints*, To Appear, 2003.

A. Fukunaga, G. Rabideau, S. Chien, and D. Yan. Toward an application framework for automated planning and scheduling. In *Proceedings of the* 15^{th} *International Joint Conference on Artificial Intelligence*, 1997.

A. Jónsson, P. Morris, N. Muscettola, K. Rajan, and B. Smith. Planning in interplanetary space: Theory and practice. In *Proceedings of the Fifth International Conference on Artificial Intelligence Planning and Scheduling*, 2000.

N. Muscettola, G. Dorais, C. Fry, R. Levinson, and C. Plaunt. Idea: Planning at the core of autonomous reactive agents. In *Proceedings of the 3d International NASA Workshop Planning and Scheduling for Space*, 2002.

N. Muscettola and P. Morris. Execution of temporal plans with uncertainty. In *Proceedings of the* 17^{th} *National Conference on Artificial Intelligence*, 2001.

P. Morris, N. Muscettola, and I. Tsamardinos. Reformulating temporal plans for efficient execution. In *Proceedings of the* 15^{th} *National Conference on Artificial Intelligence*, 1998.

N. Muscettola, P. Morris, and T. Vidal. Dynamic control of plans with temporal uncertainty. In *Proceedings of the* 17^{th} *International Joint Conference on Artificial Intelligence*, 2001.

N. Muscettola, P. Nayak, B. Pell, , and B. Williams. Remote agent: To boldly go where no ai system has gone before. *Artificial Intelligence*, 103(1-2), 1998.

D. Smith, J. Frank, and A. J/'onsson. Bridging the gap between planning and scheduling. *Knowledge Engineering Review*, 15(1), 2000.

A. Tate, B. Drabble, and R. Kirby. O-plan2: An open architecture for command, planning and control. *Intelligent Scheduling*, 1994.

D. E. Wilkins, K. L. Myers, J. D. Lowrance, and L. P. Wesley. Planning and reacting in uncertain and dynamic environments. *Journal of Experimental and Theoretical Artificial Ingelligence*, 7(1), 1995.